Algoritmos y Estructura de Datos: Examen 2 (Solución)

Grados Ing. Inf. y Mat. Inf. Diciembre 2015

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Importante:

- **Cada pregunta debe responderse en hojas diferentes**.
- Todas las hojas entregadas deben indicar **apellidos**, **nombre** y **DNI/NIE**.
- Este examen dura **100 minutos** y consta de **3 preguntas** que puntúan hasta **10 puntos**.
- Las calificaciones provisionales de este examen se publicarán el **9 de enero de 2016** en el Moodle de la asignatura junto con el procedimiento para solicitar la revisión.

(3 puntos) 1. **Se pide:** Implementar en Java el método

```
public static <E> FIFO<E> reverse (FIFO<E> q)
```

que toma como parámetro una cola FIFO `q` que no será null. Si la cola `q` es vacía entonces el método debe devolver una *nueva* cola `FIFO` vacía. En otro caso, el método `reverse` debe devolver una *nueva* cola en la que los elementos se encuentren en el orden inverso al orden de la cola `q` recibida como parámetro. Al terminar la ejecución del método, la cola `q` (si se ha modificado) debe tener los mismos elementos que tenía al empezar el método y éstos deben estar en el mismo orden. Para invertir el orden de los elementos de la cola es **obligatorio** utilizar una estructura temporal de tipo `LIFO<E>`. Los interfaces `FIFO<E>` y `LIFO<E>` están disponibles en el apéndice de este examen.

Por ejemplo, sea `q` una cola `FIFO<E>` de elementos `Integer` de tamaño 4 que contiene los elementos [0,2,5,1] en la que el '0' sería el primer elemento en ser desencolado y el '1' el último elemento en ser desencolado. El método `reverse()` debe devolver una *nueva* cola FIFO con los mismos elementos, pero en orden inverso, es decir, [1,5,2,0], donde el '1' sería el primer elemento en ser desencolado y el '0' el último. Puede asumirse que se dispone de las clases `FIFOList<E>` y `LIFOList<E>` que implementan, respectivamente, los interfaces `FIFO<E>` y `LIFO<E>`.

Solución:

1) Solución 1 (usando `for-each`:

```
public static<E> FIFO<E> reverse (FIFO<E> q) {
  LIFO<E> s = new LIFOList<E>();
  for (E e : q)  s.push(e);
  FIFO<E> r = new FIFOList<E>();
  for (E e : s)  r.enqueue(e);
  return r;
}
```

2) Usando los constructores de `FIFOList` y `LIFOList`

```
public static<E> FIFO<E> reverse (FIFO<E> q) {
  return new FIFOList<E>((new LIFOList<E>(q.toPositionList())).toPositionList());
}
```

3) Otros bucles (hay más opciones correctas que las presentadas en esta solución)

```
public static<E> FIFO<E> reverse (FIFO<E> q) {
  FIFO<E> resultado = new FIFOList<E>();
  LIFO<E> pila = new LIFOList<E> ();
```

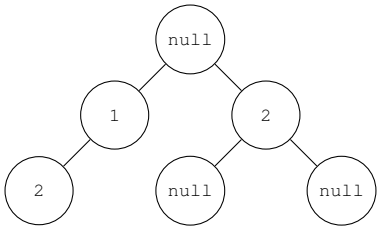
```
  Iterator<E> it = q.iterator();
  while(it.hasNext()) {
    pila.push(it.next());
  }

  while(!pila.isEmpty()) {
    resultado.enqueue(pila.top());
    pila.pop();
  }

  return resultado;
}
```

(3½ puntos) 2. **Se pide:** Implementar **de forma recursiva** en Java el método

```
public static <E> int ocurrencias(BinaryTree<E> tree, E elem)
```

que toma como parámetro un árbol binario `tree` y un elemento `elem` e indica el número de ocurrencias del elemento `elem` en el árbol `tree`. El árbol `tree` no será `null` pero podrá contener elementos `null`. Si el valor que toma el parámetro `elem` es `null` se deberán contar el número de ocurrencias de `null` en el árbol. Es **obligatorio** implementar el método **de forma recursiva**. Los interfaces `BinaryTree<E>` y `Tree<E>` están disponibles en el apéndice de este examen.

A continuación se muestra un árbol `tree` y algunos de los resultados devueltos por el método `ocurrencias`:



`ocurrencias(tree,1)` devuelve 1
`ocurrencias(tree,2)` devuelve 2
`ocurrencias(tree,null)` devuelve 3
`ocurrencias(tree,7)` devuelve 0

Solución:

1) Con el `hasLeft` y `hasRight`

```
public static <E> int ocurrencias(BinaryTree<E> tree, E elem) {
  if (tree.isEmpty()) return 0;
  return ocurrenciasRec(tree, tree.root(), elem);
}

public static <E> int ocurrenciasRec(BinaryTree<E> tree,
                                     Position<E> node,
                                     E elem) {
  int ocurrencias = 0;
  if (eqNull(elem,node.element()))
    ocurrencias ++;
  if (tree.hasLeft(node)) {
    ocurrencias += ocurrenciasRec(tree, tree.left(node), elem);
  }
  if (tree.hasRight(node)){
    ocurrencias += ocurrenciasRec(tree, tree.right(node), elem);
  }
  return ocurrencias;
}
```

```
public static <E> boolean eqNull (E o1, E o2) {
  return o1==o2 || o1!=null && o1.equals(o2);
}
```

2) Con `children`

```
public static <E> int ocurrenciasRec(BinaryTree<E> tree,
                        Position<E> node,
                        E elem) {
  int ocurrencias = 0;
  if (eqNull(elem,node.element()))
    ocurrencias ++;
  for (Position<E> v: tree.children(node)) {
    ocurrencias += ocurrenciasRec(tree, v, elem);
  }
  return ocurrencias;
}
```

3) Más compacto:

```
public static <E> int ocurrenciasRec(BinaryTree<E> tree,
                                     Position<E> node, E elem) {
  return
    (eqNull(elem,node.element()) ? 1: 0) +
    (tree.hasLeft(node)  ? ocurrenciasRec(tree, tree.left(node), elem) : 0) +
    (tree.hasRight(node) ? ocurrenciasRec(tree, tree.right(node), elem): 0);
}
```

(3½ puntos) 3. Se tiene la siguiente clase `Alumno`:

```
class Alumno {
  private Integer dni;
  private String apellidos;

  public Alumno(Integer dni, String apellidos) {
    this.dni = dni;
    this.apellidos = apellidos;
  }

  public Integer getDni() { return dni; }
  public String getApellidos() { return apellidos };
}
```

(a) Se pide: Implementar el método

```
public static Alumno [] ordenarAlumnos (Alumno [] alumnos)
```

que toma como parámetro un array de objetos de tipo `Alumno` que no será `null` y cuyos elementos tampoco serán `null` y devuelve un *nuevo* array del mismo tamaño que el array `alumnos` con los alumnos ordenados en orden creciente en función de su DNI. El método debe devolver el resultado en un nuevo array y **no debe modificar** el parámetro `alumnos`. Se recomienda utilizar una cola con prioridad usando el interfaz `PriorityQueue<K,V>` (disponible en el apéndice de este examen) para ordenar los elementos. Puede asumirse que se dispone de la clase `SortedListPriorityQueue<K,V>` que implementa el interfaz `PriorityQueue<K,V>`.

(b) Se pide: Implementar el método `compareTo` en la clase `Alumno` para que implemente el interfaz `Comparable<Alumno>` estableciendo un orden creciente de acuerdo al número del DNI. La definición de la clase alumno ahora sería:

```
class Alumno implements Comparable<Alumno> {...}
```

Sólo es necesario implementar el método `compareTo`, no es necesario copiar en la solución el resto de la clase `Alumno`. El interfaz `Comparable<E>` está disponible en el apéndice de este examen.

Apartado (a)

```
public static Alumno [] ordenarAlumnos (Alumno [] alumnos) {
  Alumno [] resultado = new Alumno [alumnos.length];
  PriorityQueue<Integer,Alumno> pq =
          new SortedListPriorityQueue<Integer,Alumno>();

  for (int i = 0; i < alumnos.length; i ++) {
    pq.insert(alumnos[i].getDni(),alumnos[i]);
  }

  int i = 0;
  while(!pq.isEmpty()) {
    resultado[i] = pq.removeMin().getValue();
    i ++;
  }

  return resultado;
}
```

Apartado (b)

```
public int compareTo(Alumno o) {
  return this.dni - o.getDni();
}
```

Interfaz `FIFO<E>`

```
public interface FIFO<E> extends Iterable<E> {
  /** Returns the number of elements. */
  public int size();
  /** Returns true if empty */
  public boolean isEmpty();
  /** Returns first element in the FIFO if non-empty, otherwise throws
   *  the exception. */
  public E first() throws EmptyFIFOException;
  /** Enqueues the element as the last element of FIFO.  */
  public void enqueue(E elem);
  /** Dequeues the first element in the FIFO if non-empty, otherwise throws
   *  the exception.  */
  public void dequeue() throws EmptyFIFOException;
  /** Returns an array with all the elements of the FIFO or an empty array
   *  of length zero if the FIFO is empty.  */
  public Object [] toArray();
  /** Returns a list with all the elements of the FIFO or an empty list if
   *  the FIFO is empty.  */
  public PositionList<E> toPositionList();
}
```

Interfaz `LIFO<E>`

```
public interface LIFO<E> extends Iterable<E> {
```

```
  /** Returns the number of elements. */
  public int size();
 /** Returns true if empty */
  public boolean isEmpty();
  /** Returns the element on top of the LIFO if non-empty, otherwise throws
   *  the exception.*/
  public E top() throws EmptyStackException;
  /** Removes the element on top of the LIFO if non-empty, otherwise throws
   *  the exception.   */
  public void pop() throws EmptyStackException;
  /** Pushes the element on top of the LIFO.   */
  public void push(E elem);
  /** Returns an array with all the elements of the LIFO or an empty array
   *  of length zero if the LIFO is empty.   */
  public Object [] toArray();
  /** Returns a list with all the elements of the LIFO or an empty list if
   *  the LIFO is empty.   */
  public PositionList<E> toPositionList();
}
```

Interfaz `Iterator<E>`

```
public interface Iterator<E> {
  public E next();
  public boolean hasNext();
  public void remove();
}
```

Interfaz `Comparable<T>`

```
public interface Comparable<T> {
  /** Returns a negative integer, zero, or a positive integer as this
      object is less than, equal to, or greater than the specified
      object o.
      El método devuelve el valor res donde:
      - res < 0 si this es menor que el objeto o
      - res = 0 si this es igual que el objeto o
      - res > 0 si this es mayor que el objeto o   */
  public int compareTo (T o);
}
```

Interfaz `PriorityQueue<K,V>`

```
public interface PriorityQueue<K,V> {
  /** Returns the number of items in the priority queue. */
  public int size();
  /** Returns whether the priority queue is empty. */
  public boolean isEmpty();
  /** Returns but does not remove an entry with minimum key. */
  public Entry<K,V> min() throws EmptyPriorityQueueException;
  /** Inserts a key-value pair and return the entry created. */
  public Entry<K,V> insert(K key, V value) throws InvalidKeyException;
  /** Removes and returns an entry with minimum key. */
  public Entry<K,V> removeMin() throws EmptyPriorityQueueException;
}
```

Interfaz `BinaryTree<E>`

```
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the left child of a node. */
  public Position<E> left(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;
  /** Returns the right child of a node. */
  public Position<E> right(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;
  /** Returns whether a node has a left child. */
  public boolean hasLeft(Position<E> v) throws InvalidPositionException;
  /** Returns whether a node has a right child. */
  public boolean hasRight(Position<E> v) throws InvalidPositionException;
}
```

Interfaz `Tree<E>`

```
public interface Tree<E> {
  /** Returns the number of nodes in the tree. */
  public int size();
  /** Returns whether the tree is empty. */
  public boolean isEmpty();
  /** Returns an iterator of the elements stored in the tree. */
  public Iterator<E> iterator();
  /** Returns an iterable collection of the the nodes. */
  public Iterable<Position<E>> positions();
  /** Replaces the element stored at a given node. */
  public E replace(Position<E> v, E e)
    throws InvalidPositionException;
  /** Returns the root of the tree. */
  public Position<E> root() throws EmptyTreeException;
  /** Returns the parent of a given node. */
  public Position<E> parent(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;
  /** Returns an iterable collection of the children of a given node. */
  public Iterable<Position<E>> children(Position<E> v)
    throws InvalidPositionException;
  /** Returns whether a given node is internal. */
  public boolean isInternal(Position<E> v)
    throws InvalidPositionException;
  /** Returns whether a given node is external. */
  public boolean isExternal(Position<E> v)
    throws InvalidPositionException;
  /** Returns whether a given node is the root of the tree. */
  public boolean isRoot(Position<E> v)
    throws InvalidPositionException;
}
```